Büchi Automata Optimisations Formalised in Isabelle/HOL

Alexander Schimpf[1,⋆] and Jan-Georg Smaus[2]

[1] Institut für Informatik, Universität Freiburg, Germany
[2] IRIT, Université de Toulouse, France
smaus@irit.fr

Abstract. In applications of automata theory, one is interested in reductions in the size of automata that preserve the recognised language. For Büchi automata, two optimisations have been proposed: bisimulation reduction, which computes equivalence classes of states and collapses them, and α-balls reduction, which collapses strongly connected components (SCCs) of an automaton that only contain one single letter as edge label. In this paper, we present a formalisation of these algorithms in Isabelle/HOL, providing a formally verified implementation.

1 Introduction

Model-checking is an important method for proving systems correct, and is applied in industrial practice [1]. In previous work [2], we present a reference implementation for an LTL (linear temporal logic) model checker for finite-state systems à la SPIN [5]. The model checker follows the well-known automata-theoretic approach. Given a finite-state program P and an LTL formula ϕ, two Büchi automata are constructed: the *system automaton* that recognises the executions of P, and the *property* or *formula automaton* expressing all potential executions that violate ϕ, respectively. Then the product of the two automata is computed and tested on-the-fly for emptiness. This implementation is realised and verified using Isabelle/HOL [7].

One important part of automata-based model checking is the translation of an LTL formula into a Büchi automaton. The standard algorithm for this problem has been proposed by Gerth *et al.* [4]. Previously to [2], we have implemented and verified this algorithm in Isabelle/HOL [8]. In this paper, we consider two of the optimisations proposed by Etessami and Holzmann [3] to reduce the size of the formula automaton.

In model checking, the system automaton is usually much larger than the property automaton, but since the size of the property automaton is a multiplicative factor of the overall complexity, it is worthwhile to put substantial effort into its optimisation [3].

The first optimisation is *bisimulation* reduction, which computes equivalence classes of states and collapses them. The algorithm of [3] uses a so-called colouring.

⋆ Supported by DFG grant CAVA, Computer Aided Verification of Automata.

1: **proc** *BasicBisimReduction*$(A) \equiv$
2: /* Init: $\forall q \in Q.\ C^{-1}(q) := 1$, and $\forall q \in F.\ C^0(q) := 1,\ \forall q \in Q \setminus F.\ C^0(q) := 2$.*/
3: $i := 0$;
4: **while** $|C^i(Q)| \neq |C^{i-1}(Q)|$ **do**
5: $i := i + 1$
6: **foreach** $q \in Q$ **do**
7: $C^i(q) := \langle C^{i-1}(q), \cup_{(q,a,q') \in \delta} \{(C^{i-1}(q'), a)\}\rangle$
8: **od**
9: Rename colour set $C^n(Q)$, with $\{1, \ldots, |C^i(Q)|\}$, using lexicogr. ordering.
10: **od**
11: $C := C^i$; **return** $A' := \langle Q' := C(Q), \delta', q'_{\mathcal{I}} := C(q_{\mathcal{I}}), F' := C(F)\rangle$;
12: /* δ' defined so that $(C(q_1), a, C(q_2)) \in \delta'$ iff $(q_1, a, q_2) \in \delta$ for $q_1, q_2 \in Q$*/

Fig. 1. Basic Bisimulation Reduction Algorithm [3]

Our formalisation has revealed that there is a mistake in the initialisation of the algorithm, which we have corrected in our implementation.

The second optimisation is *α-balls reduction*, which collapses strongly connected components (SCCs) that only contain one single letter as edge label.

The rest of the paper is organised as follows: Section 2 gives some preliminaries. Section 3 recalls the two optimisations of [3] in turn. Section 4 presents our Isabelle formalisations of those algorithms, and Sec. 5 concludes.

2 Preliminaries

We recall the basic notions of automata as used by [3]; for more details see [10].

Usually, Büchi (or finite) automata have transitions labelled with characters from an alphabet Σ. In [3], a generalisation of such labellings is considered, but for our purposes, this is not necessary and so we assume simple characters. We assume that a Büchi automaton A is given by $\langle Q, \delta, q_{\mathcal{I}}, F\rangle$. Here Q is a set of states, $\delta \subseteq (Q \times \Sigma \times Q)$ is the transition relation, $q_{\mathcal{I}} \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The language $L(A)$ is defined as the set of those ω-words which have an accepting run in A, where a *run* on word $w = a_1 a_2 \ldots$ is a sequence $q_0 q_1 q_2 \ldots$ such that $q_0 = q_{\mathcal{I}}$ and $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for all $i \geq 0$, and it is *accepting* if $q_i \in F$ for infinitely many i.

Isabelle/HOL [7] is an interactive theorem prover based on Higher-Order Logic (HOL). You can think of HOL as a combination of a functional programming language with logic. Isabelle/HOL aims at being readable by humans and thus follows the usual mathematical notation, but still the syntax needs some explanations which we provide when we come to the examples. In our presentation of Isabelle code we have stayed faithful to the sources.

3 The Original Algorithms

3.1 The Bisimulation Reduction Algorithm

Fig. 1 shows the basic bisimulation reduction algorithm in pseudo-code. The letter C with a superscript refers to the iterations of the computation of a colouring.

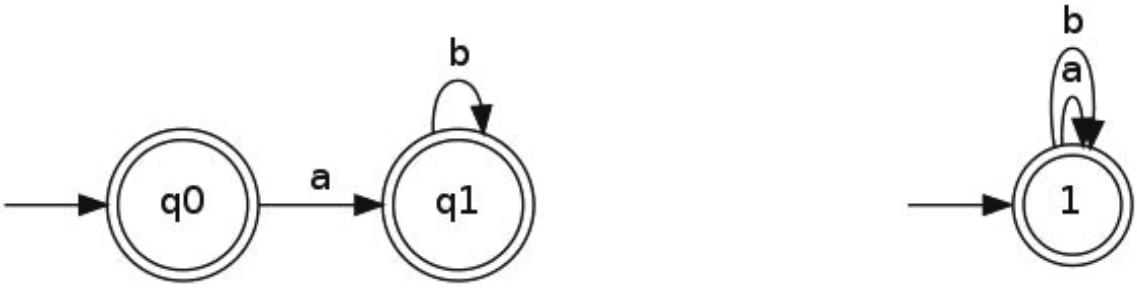

Fig. 2. An automaton and its incorrect reduction according to [3]

The idea is that in the beginning ($i = 0$) accepting states have colour 1 and non-accepting states have colour 2, and in each step, the colour of a state is obtained by its old colour and a combination of the successor state colours and the corresponding edge labels. This means that if two states have the same colour but they differ in the colours of their successors (taking into account the edge labels), then those two states must be distinguished; we say that the colouring is *refined*. In the end, states with the same colour can be joined.

The algorithm initialises not only C^0 but also C^{-1} (we might call this "pre-initialisation") which is a trick making the formulation of the algorithm more concise, by allowing for a loop condition that makes a comparison between the current and the previous colouring, even for $i = 0$.

However, our formalisation of the reduction algorithm, to be shown later, has revealed that there is a mistake in this pre-initialisation. This is illustrated in Figure 2. Here we have $Q = \{q0, q1\}, F = Q, C^0(q0) = 1, C^0(q1) = 1, C^{-1}(q0) = 1, C^{-1}(q1) = 1$. Just before the **while** we have $|C^0(Q)| = |C^{-1}(Q)| = 1$; even stronger, we have $C^0(Q) = C^{-1}(Q) = \{1\}$. What matters is that the loop condition is false and hence the loop is not entered at all. Therefore the result $C = C^0$ is computed, yielding the automaton shown on the right in Figure 2.

Our explanation is as follows: The pre-initialisation $\forall q \in Q.\ C^{-1}(q) := 1$ is conceptually wrong. It expresses that "pre-initially" ($i = -1$), there is only one colour. If by coincidence the input automaton has only accepting or only non-accepting states, then "initially" (index $i = 0$), there is also just one colour. The loop condition will then wrongly say "we have done enough refinement steps".

The problem really manifests itself for the case that $F = Q$, i.e., all states are accepting: each refinement step takes into account the edge labels and not just whether a state is accepting or not. The initialisation however only considers whether a state is accepting or not, and so not doing any refinement wrongly results in identifying all states ($q0$ and $q1$ in the example).

The conceptual mistake happens to cause no harm in the case $F = \emptyset$, since the initialisation of the algorithm establishes the property $F = \emptyset$ and trivially maintains it since no refinement is done. The accepted language is then empty.

Our solution is to replace the condition $|C^i(Q)| \neq |C^{i-1}(Q)|$ with $i \leq 0 \vee |C^i(Q)| \neq |C^{i-1}(Q)|$ (in the actual formalisation: $i > 0 \longrightarrow |C^i(Q)| \neq |C^{i-1}(Q)|$)

so that the loop body will definitely be entered for $i = 0$ at least once. This is shown in Fig. 3 and will be discussed in Sec. 4.1.

3.2 α-Balls Reduction

This optimisation may appear simple and rather specialised, but in fact, it is quite effective in our context of model checking, more precisely, on Büchi automata that are the result of a translation from *generalised* Büchi automata which in turn are the output of the formula translation [4]. Note also that the reduction does not work for finite automata; it only works for Büchi automata.

The idea of the reduction is that, if in a Büchi automaton we are ever stuck in a component, and the only transition labels in this component are α, and there is some accepting state in the component, then we can treat the entire component as a single accepting state with a self-transition labelled by α.

Definition 1. *For $\alpha \in \Sigma$, a* fixed-letter α-ball[1] *inside a Büchi automaton A is a set $Q' \subseteq Q$ of states such that:*

1. *$\alpha \in \Sigma$ is the unique letter which labels the transitions inside Q';*
2. *the nodes of Q' form an SCC of A;*
3. *there is no transition leaving Q' , i.e., no $(q', b, q) \in \delta$ where $q' \in Q'$ and $q \notin Q'$.*
4. *$Q' \cap F \neq \emptyset$.*

Proposition 1. *Given a Büchi automaton $A = \langle Q, \delta, q_{\mathcal{I}}, f\rangle$, suppose $Q' \subseteq Q$ is a fixed-letter α-ball of A. Let $A' = \langle (Q \setminus Q') \cup \{q_{new}\}, \delta', q'_{\mathcal{I}}, (F \setminus Q') \cup \{q_{new}\}\rangle$ where*

$$\delta = \{(q_1, b, q_2) \mid q_1, q_2 \in Q \setminus Q'\} \cup \{(q_1, b, q_{new}) \mid (q_1, b, q_2) \in \delta, q_1 \in Q, q_2 \in Q'\} \cup \{(q_{new}, \alpha, q_{new})\},$$

and $q'_{\mathcal{I}} = q_{new}$ if $q_{\mathcal{I}} \in Q'$, else $q'_{\mathcal{I}} = q_{\mathcal{I}}$. Then $L(A) = L(A')$.

4 Isabelle Formalisation

The work presented in this paper is a fragment of a bigger library being developed on automata in the context of model checking, in particular the construction of the *property automaton* (see Sec. 1). Modularity, generality and reuse are important concerns in this project, which is why the Isabelle code chunks presented here exhibit some aspects that we do not discuss in all detail.

Generally, automata are represented as *record types* that are parametrised by the type of the states and the type of the alphabet, among others. E.g., in line 2 in Fig. 3, $'q$ is the type of the states. The fields of these records, mostly denoted by calligraphic letters, refer to the states, the final states, etc. E.g., in line 5 in Fig. 3, $\mathcal{Q}$ gives the state set, and in line 7, $\mathcal{F}$ refers to being accepting.

[3] defines more generally: a *fixed-formula α-ball*, i.e., α is a formula.

```
 1: definition LBA_bpr_C ::
 2:       "('q, 'l, 'more) LBA_scheme ⇒ ('q ⇒ nat) nres"
 3:  where
 4:    "LBA_bpr_C 𝒜 ≡ do {
 5:       let Q = 𝒬 𝒜;
 6:       let C = (λq. 1);
 7:       let C' = (λq. if ℱ 𝒜 q then 1 else 2);
 8:       let i = 0;
 9:       (_, C', _) ←
10:       WHILEIT
11:         (LBA.LBA_bpr__whilei 𝒜 Q)
12:         (λ(C, C', i). i>0 ⟶ card (C' ` Q) ≠ card (C ` Q))
13:         (λ(C, C', i). do {
14:           let i = Suc i;
15:           let C = C';
16:           let f = (λq. (C q, ℒ 𝒜 q,  C ` successors⇘𝒜 𝒜 q));
17:           let R = f ` Q;
18:           fi ← set_enum R;
19:           let C' = fi o f;
20:           RETURN (C, C', i)
21:         }) (C, C', i);
22:       RETURN C'
23:     }"
```

Fig. 3. BPR colouring

In the formalisation we present here, we use automata that follow [4] but differ from the standard definition in one important aspect: we assume that not the *edges*, but rather the *states* of automata are labelled. We call those automata *labelled Büchi automata* (*LBA*), as opposed to *BA*. Moreover, in our representation we have a *set* of initial states rather than a unique initial state. Instead of a set of accepting states we use a predicate to express whether a state is an accepting state or not. This kind of automata representation is suitable in our context, since we formalise the algorithm in the context of the Büchi automaton construction from an LTL formula according to [4], where the output automaton corresponds to a state labelled rather than a transition labelled automaton.

Big parts of our library concern BAs, however, and technically, LBAs are defined as an extension of the BA record type where labels for the states are added and the labels for the edges are "disabled".

4.1 The Bisimulation Reduction Algorithm

The Isabelle formalisation of the algorithm from Fig. 1 is shown in Figure 3. The formalisation uses the Isabelle refinement framework [6] for writing what resembles imperative code. We explain some of the syntactic constructs.

Lines 5 to 8 accomplish the initialisation. The *let* should be understood as imperative assignment. We assign: Q is the state set of the input automaton $\mathcal{A}$; C

```
1: definition
2:    "set_enum S ≡ do {
3:      (_, m) ← FOREACHi
4:                  (set_enum__foreachi S)
5:                  S
6:                  (λx (k, m). RETURN (Suc k, m(x↦k)))
7:                  (1, empty);
8:      RETURN (λx. case m x of None ⇒ 0 | Some k ⇒ k) }"
```

Fig. 4. Numbering of sets

is the $(i-1)$th colouring and is initialised to the function that colours each state as 1; C' is the ith colouring and is initialised to the function that colours each accepting state as 1 and all others as 2. We need C and C' in order to determine whether the current iteration has actually refined the current colouring.

Line 9 performs a kind of nondeterministic assignment: the term following it is essentially a set of values, and an element of this set is assigned to $(\_, C', \_)$ nondeterministically. The refinement framework allows us to specify algorithms with such nondeterminism, prove theorems about them, and replace the nondeterminism by a deterministic implementation later and independently.

Back to the code: line 10 contains the **while**-construct in this language. Its first argument (line 11) consists of a loop invariant that must be provided by the programmer and that is used in correctness proofs. The second argument (line 12) is the loop condition corrected as explained in Sec. 3.1. The third argument (lines 13 to 21) is the loop body which takes the form of a λ-term with an abstraction over (C, C', i), applied to the argument (C, C', i) (line 21) which corresponds to the initial values explained two paragraphs above.

During each iteration, a new colouring is computed in the form of a function f that assigns a certain triple to each state; R is then defined as the image of f on Q, i.e., R is the set of all the colours of the new colouring. In order to convert those complicated triples into simple numbers, an auxiliary function shown in Fig. 4 is used; *fi*, obtained by a nondeterministic assignment as in line 9, is then the numbering of R. The new colouring is then the composition of *fi* and f. It assigns a number to each state.

The function *set_enum* computes the set of all unique numberings for a set S, so that *set_enum* S is the set of bijections between S and $\{1, \ldots, \#S\}$. The definition of the function starts with a **foreach**-construct (lines 3-7). In line 3 one possible result of the loop is chosen non-deterministically and is assigned to $(\_, m)$. In the following line a loop-invariant is provided in order to be able to prove correctness properties of the definition. The second parameter in line 5 represents the iterated set. For each element of S the body of the loop (line 6) is applied sequentially in an arbitrary order, where x is an element of S and (k, m) an intermediate result of the loop, that is propagated through the iterations starting with $(1, \mathit{empty})$. Such results correspond to a pair consisting of the next number to assign and an already constructed mapping of numbers to elements

of S. An empty mapping is denoted by *empty* and for an existing mapping m a new mapping is constructed with $m(x \mapsto k)$, that behaves like m with the exception that it maps x to k. In the last line the above constructed mapping is turned into a function: elements not in S are mapped to 0. The following lemma states the correctness of the construction:

```
lemma set_enum_correct:
  assumes "finite S"
  shows "set_enum S ≤ SPEC (λf. bij_betw f S {1..card S})"
```

The term *bij_betw f S* {1..*card S*} says "f is a bijection between S and $\{1, \ldots, \#S\}$", and the entire lemma says that the results of *set_enum S* in terms of functions f fulfil the specification "f is a bijection between S and $\{1, \ldots, \#S\}$".

Thus we have a bijection between $f(Q)$ and $C'(Q)$ in the actual procedure, i.e., line 9 in the pseudo-code in Fig. 1 is correctly implemented.

Termination After each execution of the body of the loop, the number of colours according to C' increases compared to C, or the iteration stops after that execution of that loop body. At the same time, the number of possible colours in C' is bounded by the number of states in the input automaton. Hence the number of colours cannot increase indefinitely and therefore the iteration stops eventually.

Correctness The proof of correctness of the procedure is based on the characterisation of the ith colouring. For $i > 0$, we have the following loop invariant:

```
definition (in LBA)
  "LBA_bpr_C_inv Q ≡ λ(C, C', i).
     ∀q∈Q. ∀q'∈Q. C' q = C' q'
                  ⟷ (C q = C q'
                        ∧ ℒ 𝒜 q = ℒ 𝒜 q'
                        ∧ C ‘ successors q = C ‘ successors q')"
```

This invariant corresponds exactly to the modification of C' after each iteration and is thus simple to prove based on the bijectivity of f and C'. For the general case, i.e., including $i = 0$, the following holds: whenever $C'(q) = C'(q')$ for two states $q, q' \in Q$, then either both q, q' are in F or they are both not in F.

When the iteration of the loop stops, i.e., the number of colours in C' does not change anymore compared to C, we obtain a bijection between C and C'. Considering that the invariant relates C to C', we obtain that C and C' yield the same equivalence classes, i.e., for any states $q, q' \in Q$, we have $C'(q) = C'(q')$ if and only if $C(q) = C(q')$. In Fig. 5, we have defined the characteristics that are needed in the subsequent proofs and that *LBA_bpr_C* indeed fulfils. We have chosen to introduce a definition for this characterisation because it occurs in several places in our development.

All in all, we obtain the characterisation of the resulting colouring shown by the lemma in Figure 6. Indeed, we need to assume that the set of states is

```
definition (in LBA)
  "LBA_bpr_C_char
    ≡ λC. ∀q∈Q A. ∀q'∈Q A.
            C q = C q' ⟶ (F A q ⟷ F A q')
                       ∧ L A q = L A q'
                       ∧ C ‘ successors q = C ‘ successors q'"
```

Fig. 5. Characterisation of the colouring

```
lemma (in LBA) LBA_bpr_C_correct:
  "LBA_bpr_C A ≤ SPEC LBA_bpr_C_char"
```

Fig. 6. Correctness property of the colouring

finite. Otherwise we could not apply the above shown properties about *set_enum*. Finiteness is given here by an implicit assumption denoted by "(**in** *LBA*)".

This characterisation turns out to be a sufficient condition for proving the correctness of the resulting coloured automaton. To obtain the automaton, we apply a renaming function *LBA_rename* which takes an LBA and a colouring function C and returns the LBA where each state has been renamed using the colouring function. The renaming function has properties shown in Figure 7.

For example the second line of the lemma states that the initial states of the renamed automaton are exactly the renamings of the initial states of the original automaton. The other lines state similarly that the transition function, the final states, and the labels are preserved by the renaming.

The term *inv_into Q f q* gives "the"[2] inverse element of q under f in Q. In the case that this inverse is unique, i.e., f is injective on Q, it is straightforward to show that the renaming preserves language equivalence. However, the very purpose of the colouring is to *reduce* the number of states, hence not to be injective! But even in this general case, language equivalence holds, as is expressed by the following lemma:

```
lemma (in LBA) LBA_bpr_C_rename_accept_iff:
  assumes "LBA_bpr_C_char C"
  shows "∀w. LBA_accept (LBA_rename A C) w ⟷ LBA_accept A w"
```

The proof of the lemma works constructively. The "⟵" direction consists of taking a run (sequence of states) r for word w of the input automaton and colouring r componentwise using C, i.e., r is mapped to a run $C \circ r$ in the coloured automaton.

The "⟶" direction requires an auxiliary function:

[2] This is the famous ϵ operator of HOL: It represents an arbitrary but fixed term fulfilling a given property.

```
lemma LBA_rename_simps:
  "δ (LBA_rename A C) q a = C ‘ δ A (inv_into (Q A) C q) a"
  "I (LBA_rename A C) = C ‘ I A"
  "F (LBA_rename A C) q ⟷ F A (inv_into (Q A) C q)"
  "L (LBA_rename A C) q = L A (inv_into (Q A) C q)"
```

Fig. 7. Renaming function for LBAs

Fig. 8. An LBA and its colouring

```
fun bpr_run
where
  "bpr_run A C r q0 0 = q0"
| "bpr_run A C r q0 (Suc k)
     = (SOME q'. C q' = C (inv_into (Q A) C (r (Suc k)))
                 ∧ q'∈δ_L A (bpr_run A C r q0 k))"
```

This function is needed for the following reason: if we start from a run r for w in the coloured automaton and use *inv_into Q f q* to compute a corresponding state in the input automaton for each state q in r, then these states do not necessarily "fit together", i.e., they may not form a run in the input automaton.

Example 1. Figure 8 shows an LBA accepting the word $aaa\ldots$ on the left, and the simplified automaton obtained by colouring on the right. Obviously, the state sequence in the coloured automaton is always simply a sequence of 1's. The inverse of "1" is either q_1 or q_2, so simply translating $11\ldots$ back into the original LBA would give either $q_1q_1\ldots$ or $q_2q_2\ldots$. In neither case this would correspond to a run of the original LBA.

As the example shows, constructing a run in the original LBA, given a run in the coloured LBA, is not so simple and has to be done step by step starting from the initial state. This is what the function *bpr_run* is good for. It starts by taking an inverse of the initial state of the run r of the coloured automaton, and then always picks an appropriate inverse for each next state in r. The existence of such an inverse is guaranteed by the assumption that the colouring C fulfils the characterisation according to Figure 5.

The entire procedure for constructing the coloured automaton is then given in Figure 9. By the considerations given above, this construction is correct:

```
lemma (in LBA) LBA_bpr_correct:
  "LBA_bpr A
    ≤ SPEC (λA_C. LBA A_C ∧ (∀w. LBA_accept A_C w ⟷ LBA_accept A w))"
```

```
definition LBA_bpr :: "('q, 'l, _) LBA_scheme ⇒ (nat, 'l) LBA nres"
where
  "LBA_bpr 𝒜 ≡ do {
     C ← LBA_bpr_C 𝒜;
     RETURN (LBA_rename 𝒜 C)
   }"
```

Fig. 9. Procedure for constructing the coloured automaton

In addition to language equivalence we also show that the coloured automaton is well-formed, i.e. $LBA\ \mathcal{A}_C$ holds.

4.2 α-Balls Reduction

Most of the Isabelle development on this topic uses in fact BA, not LBA, and the formalisation of α-balls reduction for LBA is based on the one for BA. However, we focus here on LBA because the use of LBA required certain adaptations that partly make the contribution of our work.

An α-ball for LBA has, of course, an Isabelle definition. However, this is not very readable and so we prefer to present the following characterisation:

```
lemma αball_L_full_def:
  "αball_L 𝒜 α Q ≡
     𝒢 𝒜 ↾ Q ∈ sccs 𝒜 ∧ Q ≠ {}
     ∧ (∀q∈Q. successors 𝒜 q ⊆ Q)
     ∧ (∀q∈Q. ℒ 𝒜 q = α)"
```

The lemma says that Q is an α-ball iff the following four conditions hold: (1) the graph of $\mathcal{A}$, restricted to Q, is an SCC of $\mathcal{A}$; (2) $Q \neq \emptyset$; (3) there is no edge leading out of Q; (4) all states in Q are labelled with α.

Fig. 10 gives an auxiliary definition for balls reduction. There is a loop for working through the SCCs. Trivial balls or balls without accepting states are removed. One-element balls are left untouched. Finally, non-trivial balls are collapsed to a single state. Based on the computation of SCCs according to Tarjan's algorithm [9] we obtain a function that computes the SCCs and then reduces, as shown in Figure 11. The ball reduction on LBAs thus returns a well-formed equivalent automaton and a set $SCCs_R$ representing its SCCs.

The correctness follows from the correctness of *LBA_reduce_ball_aux* and *tarjan*:

```
lemma (in LBA) LBA_reduce_balls_correct:
  "LBA_reduce_balls 𝒜 ≤ SPEC(λ(𝒜_R, SCCs_R).
     LBA 𝒜_R ∧ (∀w. LBA_accept 𝒜_R w ⟷ LBA_accept 𝒜 w)
     ∧ 𝒬 𝒜_R = ⋃SCCs_R
     ∧ pairwise_disjoint SCCs_R
     ∧ (∀Q∈SCCs_R. 𝒢 𝒜_R ↾ Q∈sccs 𝒜_R ∧ Q ≠ {})
   )"
```

```
definition LBA_reduce_balls_aux ::
   "[('q, 'l, 'more) LBA_scheme, 'q set set]
      ⇒ (('q, 'l, 'more) LBA_scheme × 'q set set) nres"
where
  "LBA_reduce_balls_aux 𝒜 SCCs
    ≡ FOREACHi
         (LBA_reduce_balls_aux_foreach_inv 𝒜 SCCs)
         SCCs
         (λQ (𝒜_R, SCCs_R).
           if ¬ ball_L 𝒜_R Q then RETURN (𝒜_R, {Q} ∪ SCCs_R)
           else if scc_trivial 𝒜_R (𝒢 𝒜_R ↾ Q)
                then RETURN (remove_states 𝒜_R Q, SCCs_R)
           else if ∀q∈Q. ¬ ℱ 𝒜_R q
                then RETURN (remove_states 𝒜_R Q, SCCs_R)
           else if (∃q. Q = {q}) then RETURN (𝒜_R, {Q} ∪ SCCs_R)
           else
             case ball_L_get_α 𝒜_R Q of
               None ⇒ RETURN (𝒜_R, {Q} ∪ SCCs_R)
             | Some α ⇒
                 do { (𝒜_R, Q_R) ← LBA_remove_αball 𝒜_R Q;
                      ASSERT (Q_R⊆Q ∧ Q_R ≠ {});
                      RETURN (𝒜_R, {Q_R} ∪ SCCs_R) }
           )
         (𝒜, {})"
```

Fig. 10. Auxiliary definition for α-balls reduction

```
definition LBA_reduce_balls ::
  "('q, 'l, 'more) LBA_scheme
    ⇒ (('q, 'l, 'more) LBA_scheme × 'q set set) nres"
where
  "LBA_reduce_balls 𝒜 ≡ do {
     SCCs ← tarjan 𝒜 (ℐ 𝒜);
     (𝒜_R, SCCs_R) ← LBA_reduce_balls_aux 𝒜 SCCs;
     RETURN (𝒜_R, SCCs_R)
   }"
```

Fig. 11. Ball reduction on LBAs

5 Conclusion

We have presented an Isabelle/HOL formalisation of two Büchi automata optimisations proposed by [3]. The context of this work is explained in detail in [2]: implementing full-fledged model checkers verified in Isabelle/HOL. Within this endeavour, it is worthwhile to invest effort in the optimisation of the property automaton. While the optimisations are particularly relevant for model checking, they are abstract enough to be applicable to Büchi automata in general.

The difficulty lay in making the right design decisions for the formalisation. We generally tried to model the Isabelle proofs on the paper-and-pencil proofs of the literature, but especially for the colouring, it turned out to be better to develop a new constructive proof from scratch. In that particular case, as mentioned above, the original code contained a mistake, which we discovered during our vain efforts to prove the code correct.

Concerning the first optimisation, there is also a more sophisticated algorithm [3]. However this algorithm relies on the *edge* labels in an automaton and thus it is not immediately possible to implement this algorithm in our scenario.

Our formalisation consists of around 1500 lines of code (approximately 3000 lines of code if the formalisation of Tarjan's SCC algorithm is counted).

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking, 5th print. MIT Press (2002)
2. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013)
3. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–167. Springer, Heidelberg (2000)
4. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Proceedings of the 15th International Symposium on Protocol Specification, Testing, and Verification. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman and Hall (1996)
5. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
6. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012)
7. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
8. Schimpf, A., Merz, S., Smaus, J.-G.: Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 424–439. Springer, Heidelberg (2009)
9. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
10. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B), pp. 133–192. Elsevier and MIT Press (1990)